Chapter 2
Specification PEARL Language

2.1 Extending PEARL for Distributed Systems

Since the complexity of current automation and real-time processing tasks requires the programming of distributed, fault-tolerant multiprocessor systems, the developers of PEARL have decided to extend PEARL with constructs for the programming of multiprocessors. Thus, Multiprocessor PEARL or PEARL for Distributed Systems, viz. DIN 66253, Part 3 [1], was defined as an over-layer on PEARL, and enhanced the language with constructs for the abstract descriptions of hardware and software architectures. These enabled real-time embedded systems to be co-designed in order to increase their quality of service, in particular their predictability and dependability. While not being translated into machine code, these constructs are mainly used as directives for system programs (e.g. real-time operating systems, configuration management programs, etc.) instead. Hence, Multiprocessor PEARL has further been extended in the form of a co-design methodology into the Specification PEARL language and methodology with the following properties:

- constructs to describe hardware configurations,
- constructs to describe software configurations,
- constructs to specify communication and its characteristics (peripheral and process connections, physical and logical connections, transmission protocols) as well as
- constructs to specify both conditions and methods of carrying out dynamic reconfigurations in cases of failure.

Furthermore, Specification PEARL has the following characteristics, usually required for specification languages:

- abstraction, i.e. insignificant details are suppressed, the conceptual world of the application domain is supported, and no implementation is referred to,
- application concepts and structures, relations and sequences are easily recognisable,
- easy readability, but nevertheless precise notation,

R. Gumzej, *Engineering Safe and Secure Cyber-Physical Systems*,
Studies in Computational Intelligence 632, DOI 10.1007/978-3-319-28905-2_2

- provision for unambiguous and complete descriptions of requirements and design,
- support for effective communication between clients, designers and users about the systems to be developed,
- possibility of easily extending specifications into executable prototypes,
- inclusion of appropriate real-time executive and dynamic re-configuration management programs and
- systematic integration of the specification method into the entire development process.

Specification PEARL extends PEARL for distributed systems to enable the specification of asymmetrical architectures as well as towards a more distinctive description of systems' communication interfaces and intelligent peripheral devices. Along with the textual man-readable specification language, graphical constructs with the same properties have been defined as basis for an appropriate CASE environment. Within the environment behavioural modelling (of program tasks) by timed state transition diagrams is supplemented. The output models (virtual machines), representing the target systems' hardware and software architectures as well as application program prototypes are subject to verification and validation. As they can be checked for correctness, consistency and coherency, the methodology provides a verification phase preceding the validation phase, where a system's coherence with the prescribed functional, temporal as well as safety and security requirements is checked.

Herewith, a methodology is defined, enabling systematic design of the structure as well as the behaviour of the designed system. Its benefits are the standard-based user-readable syntax, which can serve as input of compilers, configuration managers or loaders, the ability to model a system's dynamic behaviour, which is suitable for validation by simulation, and summarising all this, the ability to check a system's feasibility before implementing it. The methodology is presented in detail in the next chapter.

In the sequel, the Specification PEARL language is presented, followed by the description of its associated CASE environment with its program libraries, to be used in the design, verification, validation and deployment phases. The modelling technique based on timed state transition diagrams is presented to demonstrate, how program tasks are formed. Finally, an example of a typical usage scenario and an existing prototype of a distributed hard real-time system [2] is considered as a case study.

2.2 Specification PEARL Notation

A system architecture specification consists of DIVISIONs, which describe different associated layers of the system design in considerable detail (e.g. Fig. 2.1):

STATION	processing node(s) hardware description,
CONFIGURATION	software unit(s) description,
NET	network interconnection(s) description,

```
ARCHITECTURE;
STATIONS;
 NAMES: KP;
   PROCTYPE: MC68370  AT 20 MHz;
   WORKSTORE: SIZE 65536  SPACE 0 - 'FFFF'B4
      READ/WRITE  WAITCYCLES 1;
    WORKSTORE: SIZE 32768  SPACE 0 - '7FFF'B4
      READONLY  WAITCYCLES 1;
    INTERFACE: KP_IO (DRIVER: KPINOUT;
      DIRECTION: INPUT; SPEED:20971520 BPS;
      UNIT:FIXED);
   STATEID: (NORMAL, CRITICAL);
   STATIONTYPE: KERNEL;
   SCHEDULING: EDF;
    MAXTASKS: 20;
    MAXSEMA: 5;
   MAXEVENT: 15;
   MAXEVENTQ: 5;
   MAXSCHED:30;
   TICK: 1E-3 SEC;...


   SYSTEM;
    NAMES: KP;
        KP.KP_IO INOUT;
       NAMES: Sensor 1;
         Sensor1.S1 OUT;
       NAMES: Sensor 2; ...
       NAMES: TP1;
         TP1.S1 IN;
         TP1.TP1_IO INOUT;
       NAMES:TP2; ...
   SYSEND;
```

```
CONFIGURATION;
  COLLECTION KP_WS;
    PORTS KP_TP1-lin, KP_TP2-lin;
    CONNECT KP_WS.KP-TP1_lin INOUT TP1-WS.TP1_KP_lin
      VIA KP.KP_IO;
  CONNECT KP_WS.KP_TP2_lin INOUT TP2_WS.TP2_KP_lin
      VIA KP.KP_IO;
  COLEND;

COLLECTION TP_WS;
  PORTS S1, TP1_KP_lin;
  CONNECT TP1_WS.S1 IN VIA TP1.S1;
  CONNCET-TP1_WS.TP1_KP_lin INOUT
    KP_WS.KP_TP1_lin VIA TP1.TP1_IO;

  MODULES TP1_WS_M1;
    EXPORTS(Side 1);
    TASK Side1
      TRIGGER PORT S1;
      DEADLINE 100;
    TASKEND;
  MODEND;
COLEND;...

CONFEND;
ARCHEND;

  NET;
    KP.KP_IO <-> TP1.TP1_IO;
    KP.KP_IO <-> TP2.TP2_IO;
    TP1.TP1_IO<-> Sensor1.S1;
    TP2.TP2_IO<-> Sensor2.S2;
  NETEND;
```

Fig. 2.1 An example of a textual architecture description expressed in specification PEARL

SYSTEM SW/HW	interface(s) description and
PERIPHERAL	intelligent peripheral device(s) description.

Since contemporary specification formalisms use graphical notations, graphical constructs with the same semantics as the textual BNF-based descriptions were defined as basis for the associated CASE environment. An overview of Specification PEARL's textual syntax and graphical notation is given in Appendix A and Appendix B, respectively.

2.2.1 Hardware Configuration

In the STATION division a system's processing nodes (*stations*) are introduced stating their most important characteristics. Stations are treated as black boxes with connections through their interfaces. To allow for multiprocessor nodes, a *composite station* is defined to be a set of stations, which are logically and physically strongly connected (i.e. they share the same housing or at least the same connections with other stations and/or intelligent peripheral devices). The basic components of a station are its processors (*proctypes*), working storages (*workstores*) and different types of *devices*. There may be multiple stations in a system, so each one of them is uniquely identified. Each station in a system maintains its *state* information.

There are several types of stations. The default type is the BASIC station, which stands for a general-purpose processing node. To be able to describe asymmetrical architectures, two additional types of processing nodes have been defined: TASK for pure application task execution and KERNEL for real-time operating system execution. Since for CPS intelligent peripheral devices are very important, the PERIPHERAL station type was introduced to represent this kind of stations.

Besides the before-mentioned general attributes, stations may also have additional attributes depending on the station's type. A multiprocessor node is characterised by the "PART OF" attributes of its constituent processing nodes. Kernel stations have properties, which are specific to them and are relevant to software designers (e.g. scheduling strategy, maximum number of active tasks, maximum number of synchronisers, events, queued events and schedules supported, real-time clock resolution etc.).

Processor's (PROCTYPE) properties are its unique ID and speed descriptor, which indicates the clock generators' frequency. Although this information may seem irrelevant at this point, one may choose to drive processors with different frequencies, which affects their processing speed. Hence, for a profiler or schedulability analyser this information is crucial to estimate the actual execution times of the individual instructions and tasks.

Work-stores (WORKSTORE) are described by their capacities and memory maps (showing the purpose of different memory areas). The wait-cycles, associated with the individual work-store areas, may also be specified (on-chip, random access or read-only memories usually have different access times). This information is used by compilers to determine the maximum execution times of tasks, being loaded to these memory areas, or the time required to access their data during execution.

Devices (DEVICE) are identified by IDs (like stations, but they may be assigned a logical name for easier reference). The device types may vary and have different attributes assigned depending on their nature. Currently, INTERFACE, TIMER and SHARED variable device types are foreseen. The use of standard devices is supported by the generic device specification.

Any net topology of a distributed system can be described by point-to-point connections. Hence, a NET division describes the physical connections between the stations of a system by their logical names and directions.

A SYSTEM division encapsulates a hardware description and the assignment of all relevant symbolic names to hardware devices, where the described components from the station and net divisions are referenced by their IDs.

A PERIPHERAL division provides the details about the intelligent peripheral devices attached to a system. The peripheral devices are identified by their IDs. Their connections to the stations of the system are described by the logical names of the interfaces, they are attached to, as well as the attributes of the interfaces used for communication (e.g. direction of data flow, protocol used and any additional signals which may be necessary for communication). To support schedulability analysis, every signal from a peripheral device can be associated with its minimum inter-arrival time.

2.2.2 Software Configuration

The CONFIGURATION division deals with a system's software architecture. The biggest program part associated with a STATION is a COLLECTION. Collections are composed of modules (MODULE) of tasks (TASK) which communicate through the respective collection's ports (PORT). Each program part has its unique name for reference.

Modules are mainly meant for information hiding and sharing among bigger collections' parts. Hence, they are further described by their IMPORTS and EXPORTS, where it is stated which data structures and tasks are shared with other modules.

Tasks are described by their trigger conditions and response times. Task alternatives may be provided in order to increase fault tolerance by graceful degradation through task scheduling—an alternative with shorter run-time or longer response time may be scheduled to maintain a schedule's feasibility.

Collections are software architecture units, being associated with stations. They are loaded to stations by re-configuration management programs, being in charge of the initial loading and starting the initial tasks. It is possible to specify under which conditions certain collections are to be removed from a station and which collections are to be loaded instead. The initial configuration as well as reconfiguration conditions are station state dependent. The latter is maintained by the already mentioned reconfiguration management programs or *configuration managers*.

Collection's ports are used for inter-collection (inter-station) communication. Hence, they are associated with appropriate station interfaces. The connections between the ports of collections are described by their directions and line attributes. Port's line attributes state which connections are always used (VIA attribute) and which ones can be chosen from a preference list based on the PREFER attribute (e.g. when using multiple different interfaces for the same communication line to increase line robustness).

2.3 Specification PEARL CASE Environment and its Program Libraries

Most of the design methodologies do not consider the target platform—hardware architecture and operating system—and only some of them have their associated CASE environments. Those, which do and produce executable code, use off-the-shelf operating systems with the corresponding tools being strictly bound to the target environments, from which they also inherit their strengths and weaknesses, in particular limitations in their capabilities, connectivity and real-time ability.

From this point of view, it was meaningful to apply a holistic approach also to building the Specification PEARL CASE environment. Since the PEARL language already includes appropriate calls to the operating system, an appropriate real-time operating system was developed for it. To provide for the appropriate parameterisation

of the operating system and appropriate consideration of the system architecture and interfaces a hardware abstraction layer was built in form of a basic executive and data interchange program. Two libraries were built for Specification PEARL and bundled with its CASE environment [3].

The first program library represents the real-time operating system, which can be easily combined with the designed application code. A rich set of system calls supporting real-time operation was foreseen, based on past experience with the RTOS-UH [4], which was primarily bound with the PEARL90 [5] compiler to build and execute real-time applications on a proprietary platform. The HaRTOS [2] real-time operating system supports the PEARL's tasking model and system calls as well as the deadline-driven scheduling strategy and is thus capable of ensuring hard real-time operation. It is programmed in C and can, thus, be compiled for any hardware platform. HaRTOS resources are pre-determined (e.g. maximum number of tasks, synchronisers, signals, events or queued events) by adequately configuring the station's parameters.

The second program library—the Configuration Manager (CM)—is meant to be used as the main executive program at each station. It represents a hardware abstraction layer that is, as configured by a hardware architecture model, mainly used to define the structure and interfaces of each station. It initiates the execution of the initial task and monitors the station state. In case the station state changes, it performs the pre-specified re-configuration actions, which mainly comprise fail-safe ending of the current execution and fast scenario switching to another (collection of) task(s). HaRTOS, is accessible through a proprietary CM communication channel. The rest of the station's communication channels are used as pre-configured by the hardware architecture description through appropriate CM data interchange methods.

In the Specification PEARL CASE environment, hardware and software architectures are designed conjunctly—one may start the design from either point of view and associate them later on. A completeness and consistency check is done in order to ensure model completeness and parameter consistency. The Specification PEARL CASE environment, which encompasses modelling and co-simulation tools, also enables cross-development for the specified hardware architectures by cross-compilation of its target platform models. The environment supports the Specification PEARL project life-cycle (c.p. Fig. 3.1), as presented in the next chapter.

2.3.1 Configuration Manager

At each processing node (station) execution starts by initiating its configuration manager (CM) which, in turn, loads the initial collection by triggering the latter's initialisation tasks. In stations without a real-time operating system, the collection's main task is started and delegated control to by the CM, whereas otherwise the CM acts as a front-end to the operating system, and uses appropriate system calls and system ports to transfer system requests to/from RTOS-enabled nodes to schedule the collection's tasks.

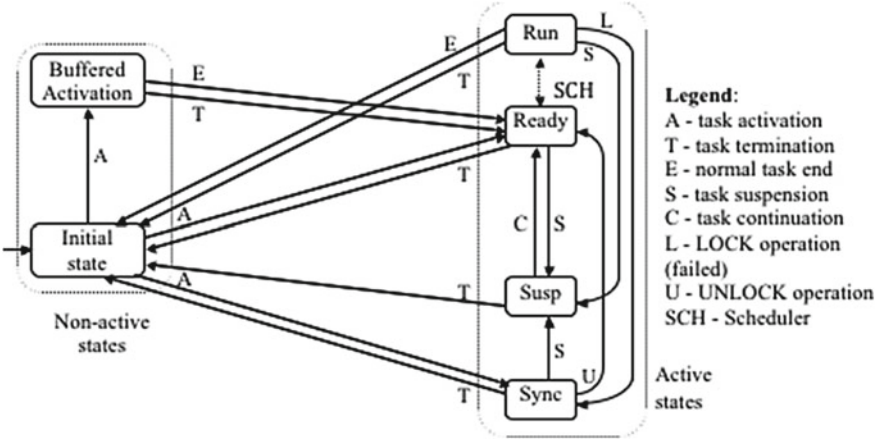

Fig. 2.2 Task model of RTOS

Besides local execution, the CM is also responsible for communication with other stations, and for co-operation among the tasks of the same collection. Hence, it must establish port-to-port connections through the interfaces of the station and provide for task synchronisation through HaRTOS. Synchronisation and system service requests are serviced on the same station, in case the station is configured to run the operating system. Otherwise, these requests are delegated to the corresponding (KERNEL) station through a proprietary port. The functions of the CM are described in detail in Appendix C.

2.3.2 Operating System

The implementation of the HaRTOS Real-Time Operating System (HaRTOS) is primarily oriented at PEARL. It supports PEARL's task model (cp. Fig. 2.2) and the system calls defined by standard PEARL with a few enhancements. The detailed description of the RTOS library's API and of its system services is given in Appendix D. In part, the implementation of HaRTOS also addresses the already mentioned CM.

According to Fig. 2.2, tasks may be active or inactive (dormant). They are activated by a (scheduled) activation (A), and return to inactive state after a normal task end (E) or task termination (T). When active, tasks may be running or awaiting their *run* conditions fulfilment, in the queue of *ready* tasks or in one of the suspended states. Ready tasks are ready to run, however, they are awaiting their turn—their queue is maintained by the HaRTOS scheduler (SCH). Tasks may be suspended for two reasons—awaiting resource allocation or synchronisation. The tasks waiting for resources are put into the *Susp* state by task suspension (S) operations and are

"awakened" by task continuation (C) operations as soon as the resources they await become available. Usually, suspended tasks are scheduled to wait for an interrupt or signal. Tasks can be suspended for synchronisation (*Sync* state) by locking (L) their presence with a semaphore and unlocking them (U) as soon as the synchronisation conditions are fulfilled. Suspended tasks becoming ready are again placed into the queue of ready tasks awaiting execution. If an already active task is activated, which may occur in case a corresponding event is triggered before the task is ended or terminated, its activation request is buffered until the task is inactivated. After that a buffered task activation may be performed. Since this operation may cause "race-conditions", it is sensible to check activation pre-conditions before saving tasks in the activation buffer. If a buffered activation originates from the same event that activated the currently active task, the event is already being handled and the new activation should not be stored. Otherwise, a new activation is sensible and should be allotted a new context.

2.4 Specification PEARL Behavioural Model

The program tasks of an application represent the processes in a running system. They are mainly characterised by their activation conditions and timing limitations as well as by their adherence to collections and modules. This information is sufficient to build a coarse program model, but it is not enough to determine its feasibility. Therefore, timed state transition diagrams (TSTD) were introduced to represent them [6]. Their synchronisation and inter-communication are realised by calls to the configuration manager and the real-time operating system of the station, respectively.

TSTD are hierarchical finite state automata consisting of

- start states: task activation conditions and initialisation actions,
- transient states: atomic activities with possibly predictable duration,
- super-states: non-atomic activities—hierarchical decomposition of working states, and
- final states: finalisation actions.

The connections between states represent the progress of tasks from start to final states. In each state they spend some time, so their progression through them also represents their progression in time. All connections among states are local (i.e. bound to one task). In every state some actions, whose execution is considered atomic, may be executed. These actions also trigger the continuation pre-conditions of the states. Intertask co-operation is enabled—by appropriate system calls to the operating system through the configuration manager. Operating system and configuration manager are accessible through their APIs and behave in concordance with the pre-specified system configuration.

The formal representation of a program configuration is the union of the tasks' models:

$$M^* = \cup_{i=0}^{n} M_i$$

Any task is represented by an eight-tuple:

$$M_i = (S, \Sigma, V, I, \delta, \varepsilon, \tau, E)$$

with the following components:

1. S_i: set of start states (there may be more, since there may be multiple trigger conditions for a single task),
2. Σ_i: set of input symbols (states trigger conditions),
3. V_i: set of all states,
4. I_i: set of time intervals (every state may be associated with one; $i_i = (v_i, t_i)$ where for each $i_i \in I_i$, $v_i \in V_i$ is the current state, $t_i \in I_i$ is the predefined time-out interval),
5. δ_i: state transition function,
6. ε_i: semantic state function (represents actions which change the internal state of the station and enable inter-task synchronisation and communication),
7. τ_i: transition function in case of time-out,
8. E_i: set of final states.

Any state is characterised by the following data:

- state type (start, transient, supet or final state),
- pre-condition for the state's execution (trigger condition for a start state),
- time-out condition (shortest, mean and maximum execution time),
- time-out action (state to which execution is diverted in case the time-out occurs),
- connection to the next state(s) in case the continuation condition(s) are fulfilled on-time and task execution within the state is successful,
- activities carried out within the framework of this state (designer-defined actions and system calls).

Start states represent different task entry points. Transient states represent states within the execution of a task where a task resides and does some action(s) awaiting the pre-condition(s) of its successor state(s) to be fulfilled. Only initial tasks' start states have the possibility of explicit (on-demand) activation. To enter other task states the following types of pre-conditions must be fulfilled:

- external events: int(number), representing interrupts (discrete signals),
- internal events: sig(identifier), representing signals,
- timers: timer(at, every, during), representing timer signals,
- general conditions: cd(expression), i.e. expressions returning Boolean results from the evaluation of internal station/program states or data structures of the operating system.

Task execution progresses from state to state upon fulfilment of a pre-condition of a successor state. Upon successful completion of the execution of the task finalisation actions within its final state control is returned to its initial/start state(s). Upon fulfiling

the pre-condition of a super-state, control is automatically transferred to the start state of its sub-model. When the final state of the sub-model is reached, control is returned to the super-state awaiting continuation pre-conditions.

Any transient state may be allotted a minimum (*minT*) and maximum (*maxT*) time-frame for its execution. The time-out condition is set to the maximum time-frame at the beginning of each state's execution. If the time-out condition is not reached before the condition to proceed to the next state is fulfilled, the corresponding connection for successful continuation is followed. If a minimum time-frame is foreseen, the continuation conditions are not checked before the specified time has elapsed. In case a time-out occurs, an appropriate on-timeout action is executed. If a time-out occurs and no on-timeout action is specified, an error is raised (and logged in the co-simulation).

The activities within a state are a set of actions, which are carried out while the task is in this state. It is assumed that the actions form a single block of program statements including system calls to the operating system and/or configuration manager, around which the control structure is formed while transforming the state chart to program code. The designer's estimates of their minimum and maximum execution times are the basis for setting the respective time-frames for the state.

The system calls, performed within a state, may change station state and hereby affect the execution of the current or another task. They may claim and release resources. They may make inquiries on the internal state of the station or change the internal state of the station. They may synchronise task execution. They may transfer task data to another task/collection by utilising appropriate system calls to the CM with references to appropriate ports and interfaces as pre-configured by the hardware/software architecture.

Hence, we may conclude that, although task state transition diagrams determine the course of individual task execution, the overall task execution is controlled by the operating system and the CM. In case a context switch is necessary, the current task's state is saved with the task's context and re-established, when task execution continues.

2.4.1 Task-Forming Rules

The role of a "task" is the same in the Specification PEARL methodology as it is in the associated programming language PEARL [7, 8]—any procedure, which needs to be carried out within a given time-frame, is a task. Therefore, we can generally say that a task is the greatest program unit, to which a maximum execution time or a deadline can be assigned.

The problem in trying to break task operations down into states is that simple tasks have just three states, viz. start, working (transient) and final. New states are only introduced (1) if a time-limited atomic (sub) operation is identified, (2) if synchronisation or communication between tasks is necessary or (3) to define branching into different continuation paths depending on the pre-conditions of successor states. The following criteria were selected to form task states:

- a state represents a single logical activity, which is only dependent on its pre-conditions and whose execution time can be determined or predicted,
- any task must have at least one start state and one or more transient-/super- and final states,
- to facilitate good decomposition, a complex state shall be broken down into simpler states by introducing a super-state and defining its state-transitions in a sub-diagram.

2.4.2 Translation from Timed State Transition Diagrams to Program Tasks

When deploying the system model, program code is automatically generated from TSTD diagrams. The general form of task prototypes, obtained from task TSTDs, is shown in Fig. 2.3.

TSTD task models are translated to program tasks in two forms:

1. target-platform-oriented, as they can be compiled by a corresponding compiler and executed on the specified hardware architecture, and
2. simulation-oriented, as they can be used and interpreted by co-simulation in a simulation environment.

The main difference between the two forms is the way external events are handled. In the first case, they are generated by the environment and handled as hardware interrupts (by appropriate device drivers), whereas in the second one, they are generated in the co-simulation environment and handled as software signals (by stub device drivers). In both cases, they are handled by the station's CM and operating system.

2.5 Case Study—Railroad Crossing

This is a well-known example used throughout the literature on formal methods for the domain of real-time systems. It is a good example to demonstrate the principles of the methodology described in this book. Here, we consider a crossing with two tracks, although there could be more. There are two sensors, S1 and S2, guarding the railroad crossing one on each side (Fig. 2.4). The railway barriers are closed and opened upon receipt of the corresponding signals.

For this application, the following demands and restrictions hold:

- Safety: if a train is in the crossing, the railway barrier is closed,
- Responsiveness: the railway barriers are open most of the time,

```
MODULE module_name;
SYSTEM;
! interrupts, signals and system variables definitions
PROBLEM;

task_name : PROCEDURE (state_id REF INT);
  DCL timeout BIT;
                                /* initialisation of all global structures
  timeout:='0'B1; ! timeout indicator
  WHILE '1'B1 REPEAT
    CASE state_id
      ALT (0) ! START state:
        IF timeout EQ '1'B1 THEN
                                    /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
                                /* RESUME task after fulfillment of the
          timeout:='1'B1;
          DELAY maxT;
                                    /* perform the appropriate start states
                /* check if any of the next working / super / end states'
                            /* if they are fulfilled, set timeout to false and
          NEXT state_id;
        FIN;
      ALT (1) ! for a WORKING state:
        IF timeout EQ '1'B1 THEN
                                    /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
          DELAY minT;
          timeout:='1'B1;
          DELAY maxT;
                                        /* perform Action=statements;
                /* check if any of the next working / super / end states'
                            /* if they are fulfilled, set timeout to false and
          NEXT state_id;
        FIN;
      ALT (2) ! for a SUPER state:
        IF timeout EQ '1'B1 THEN
                                    /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
                            /* set the state_id variable to the start state of
          NEXT state_id;
        FIN;
      ALT (3) ! for a SUPER state (as addition) - "return to" state:
                                /* set the state_id variable to the next state
              /* for a SUPER state (the sub-diagram states numbered
          NEXT state_id;
      ALT (n) ! END state:
          DELAY minT;
                                        /* perform Action=statements;
          state_id:=0;
                            /* reset task state (state_id = super_state + 1)
          NEXT state_id;
          timeout:='0'B1; ! reset timeout indicator
    FIN;
  END;
END;

MODEND;
```

```
#include "module_name.h"

/* interrupts, signals and system variables definitions */

void task_name(int &state_id;) {
  bool timeout;
*/
  timeout=false; /* timeout indicator */
  while (1) {
    switch (state_id) {
      case 0: { /* START state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
trigger conditions; */
          timeout=true;
          Delay(maxT);
Action=statements; */
pre-conditions are fulfilled; */
set the state_id variable accordingly; */
          Next(state_id);
        } }
      case 1: { /* for a WORKING state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
          Delay(minT);
          timeout=true;
          Delay(maxT);
*/
pre-conditions are fulfilled; */
set the state_id variable accordingly; */
          Next(state_id);
        } }
      case 2: { /* for a SUPER state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
the super state's sub-diagram; */
          Next(state_id);
        } }
      case 3: { /* for a SUPER state (as addition) - "return to" state: */
*/
consecutively) */
          Next(state_id); }
      case n: { /* END state: */
        Delay(minT);
*/
          state_id=0;
in case of a return from a sub-diagram */
          Next(state_id);
          timeout=false; /* reset timeout indicator */ }
    }
  }
}
```

* DELAY and NEXT are there for simulator control. DELAY represents a "busy wait"' while NEXT instructs the simulator to dispatch ("preemption point").

Fig. 2.3 Representations of a TSTD in PEARL and annotated C

under the following pre-conditions:

- a train shall not arrive (in the protected area) before the previous one has left,
- if a train arrives from the left, it leaves on the right and vice-versa,
- $dmin$ is the minimum time for a train to arrive in the railroad crossing after the sensor detects it,
- $dopen$, $dclose$ are maximum opening/closing times of the railway barrier.

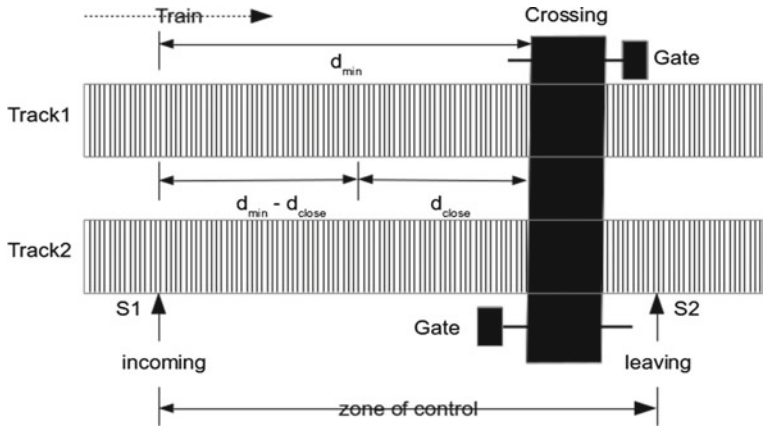

Fig. 2.4 Scheme of the railroad crossing problem

An algorithmic solution to this problem reads as follows:

var x ranges over Tracks;
initial values:

Deadline(x) := ∞ **forall** x $\in$ Tracks Dir := open;

forall x in parallel **repeat**
block

if TrackStatus(x) = coming **and** Deadline = ∞ **then**
Deadline(x) := $dclose + dmin$;
endif
if TrackStatus(x) = empty **and** Deadline < ∞ **then**
Deadline := ∞;
endif
if Dir = open **and not**(SafeToOpen) **then**
Dir := close;
endif
if Dir = close **and** SafeToOpen **then**
Dir := open;
endif

endblock

The solution of the problem is represented by an asymmetrical system with three stations (Table 2.1). It consists of a control node (KP running the operating system) and two task processing nodes (TP1 and TP2) servicing the signals coming from two sensors—one at each side of the gate. The task processors are linked to the control node and accept input signals from their respective sensors Sensor 1 (S1) and Sensor 2 (S2). The sensors S1 and S2 have to be activated one after the other for an adequate passing of a train.

Table 2.1 System architecture model in Specification PEARL and internal representation

ARCHITECTURE:	[Station types]
STATIONS:	COMPOSITE = 0
NAMES: KP;	BASIC = 1
STATEID: (NORMAL);	TASK = 2
STATIONTYPE: KERNEL;	KERNEL = 3
NAMES: Sensor1;	PERIPHERAL = 4
STATEID: (NORMAL);	[Stations]
STATIONTYPE: PERIPHERAL;	KP = KERNEL
NAMES: Sensor2;	TP1 = TASK
STATEID: (NORMAL);	TP2 = TASK
STATIONTYPE: PERIPHERAL;	Sensor1 = PERIPHERAL
	Sensor2 = PERIPHERAL
NAMES: TP1;	[KP]
STATEID: (NORMAL);	Step = 10
STATIONTYPE: TASK;	[TP1]
NAMES: TP2;	Step = 10
STATEID: (NORMAL);	Supervisor = KP
STATIONTYPE: TASK;	[Collections.TP1]
STAEND;	Name1 = TP1_WS
NET:	
KP.KP_TP1_lin INOUT TP1.TP1_KP_lin;	[TP1_WS]
KP.KP_TP2_lin INOUT TP2.TP2_KP_lin;	State = NORMAL
TP1.S1 IN;	[Tasks.TP1_WS]
TP2.S2 IN;	Name1 = Sensor1
TP1.TP1_KP_lin INOUT KP.KP_TP1_lin;	[Sensor1]
TP2.TP2_KP_lin INOUT KP.KP_TP2_lin;	Trigger = PORT S1
NETEND;	Deadline = d_Sensor
SYSTEM:	Init = Sensor1.ini
NAMES: KP;	[TP2]
KP.KP_TP1_lin INOUT TP1.TP1_KP_lin;	Step = 10
KP.KP_TP2_lin INOUT TP2.TP2_KP_lin;	Supervisor = KP
NAMES: Sensor1;	[Collections.TP2]
NAMES: Sensor2;	Name1 = TP2_WS
NAMES: TP1;	[TP2_WS]
TP1.S1 IN;	State = NORMAL
TP1.TP1_KP_lin INOUT KP.KP_TP1_lin;	[Tasks.TP2_WS]
NAMES: TP2;	Name1 = Sensor2
TP2.S2 IN;	[Sensor2]
TP2.TP2_KP_lin INOUT KP.KP_TP2_lin;	Trigger = PORT S2
SYSEND;	Deadline = d_Sensor
CONFIGURATION:	Init = Sensor2.ini

(continued)

Table 2.1 (continued)

COLLECTION KP_WS	[Sensor1]
PORTS KP_TP1_lin,KP_TP2_lin;	Step = 1
CONNECT KP_WS.KP_TP1_lin INOUT TP1_WS.TP1_KP_lin VIA	[Collections.Sensor1]
KP.KP_TP1_lin;	Name1 = Sensor1_WS
CONNECT KP_WS.KP_TP2_lin INOUT TP2_WS.TP2_KP_lin VIA	[Sensor2]
KP.KP_TP2_lin;	Step = 1
COLLECTION TP1_WS	[Collections.Sensor2]
MODULES	Name1 = Sensor2_WS
TP1_WS_M1	[Port Map]
EXPORTS (Sensor1)	Port1 = KP.KP_TP1_lin< − >TP1.TP1_KP_lin
TASKS	Port2 = KP.KP_TP2_lin< − >TP2.TP2_KP_lin
Sensor1 (TRIGGER PORT S1,DEADLINE d_Sensor)	Port3 = Sensor1.S1< − >TP1.S1
PORTS S1,TP1_KP_lin;	Port4 = Sensor2.S2< − >TP2.S2
CONNECT TP1_WS.S1 IN VIA TP1.S1;	
CONNECT TP1_WS.TP1_KP_lin INOUT KP_WS.KP_TP1_lin VIA	
TP1.TP1_KP_lin;	
COLLECTION TP2_WS;	
MODULES	
TP2_WS_M1	
EXPORTS (Sensor2)	
TASKS	
Sensor2 (TRIGGER PORT S2,DEADLINE d_Sensor)	
PORTS S2,TP2_KP_lin;	
CONNECT TP2_WS.S2 IN VIA TP2.S2;	
CONNECT TP2_WS.TP2_KP_lin INOUT KP_WS.KP_TP2_lin VIA	
TP2.TP2_KP_lin;	
CONFEND;	
PERIPHERALS:	
NAME: Sensor1;	
INTERMESSAGE PERIOD: 10 MICROSEC;	
NAME: Sensor2;	
INTERMESSAGE PERIOD: 10 MICROSEC;	
PERIPHEND;	
ARCHEND;	

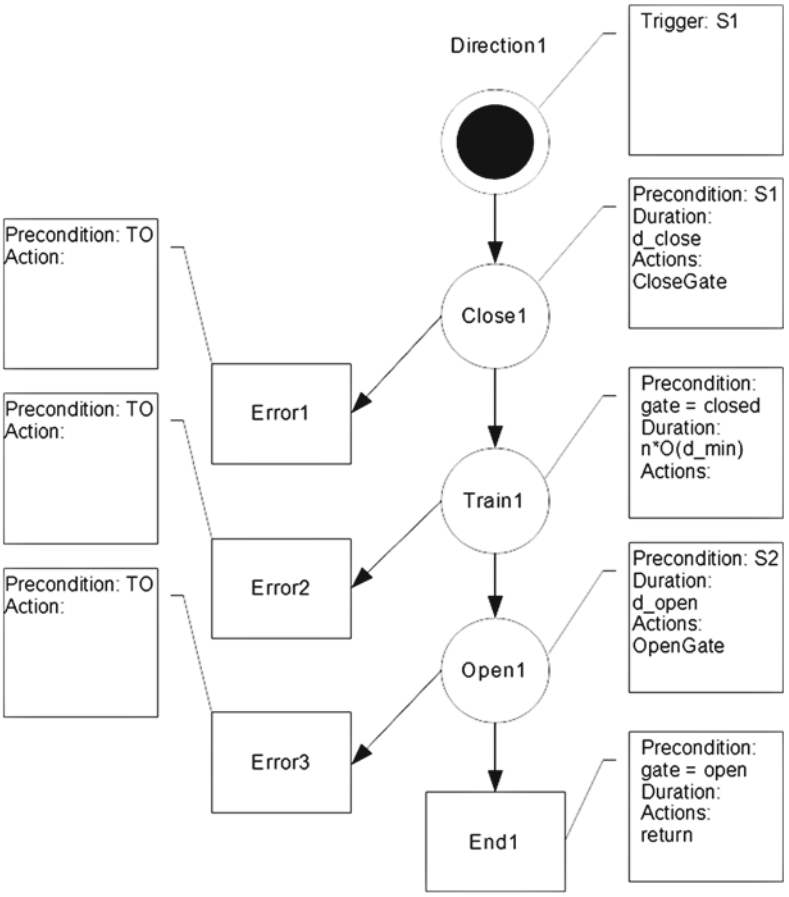

Fig. 2.5 The TSTD to program solution for task Direction1

Table 2.1 shows the textual description in Specification PEARL syntax on the left and the internal interpretation of the architecture description for the example on the right-hand side. Figure 2.5 and Table 2.2 show the TSTD diagram and the internal representation of the task handling a one-way passage of a train—from S1 to S2. The other way around is the same—only the order, in which the sensors send their signals, changes.

The internal representation of the system's architectural elements (e.g. Tables 2.1 and 2.2) is used for storing architecture data in a uniform manner for all model elements. The additional properties and TSTD code fragments are stored separately in the element database. The internal representations are used within the associated CASE environment to support the life-cycle of Specification PEARL projects—to be able to create, update, simulate and deploy the designed models. The layout of Specification PEARL projects is further described in Appendix E.

Table 2.2 Internal representation of TSTD Direction1

[State types]
START = 0
TRANSIENT = 1
END = 2
[States]
Sensor1 = START
Close1 = TRANSIENT
Train1 = TRANSIENT
Open1 = TRANSIENT
Error1 = END
Error2 = END
Error3 = END
End1 = END
[Sensor1]
Precondition = S1
MinT = 0
MaxT = 0
Next = Close1;
Action =
[Close1]
Precondition =
minT = 0
maxT = d_close
Next = Train1; Error1;
NextTO = Error1
Action = REQUEST SEMA1;
[Train1]
Precondition = NOT TRY SEMA1
minT = 0
maxT = n*d_min
Next = Open1; Error2;
NextTO = Error2
Action =
[Open1]
Precondition = S2
minT = 0
maxT = d_open
Next = End1; Error3;
NextTO = Error3
Action = RELEASE SEMA1;
[Error1]

(continued)

Table 2.2 (continued)

Precondition = TO
Action =
[Error2]
Precondition = TO
Action =
[Error3]
Precondition = TO
Action =
[End1]
Precondition = TRY SEMA1;
Action =

References

1. 66253 Part 3: Pearl for distributed systems. Technical Report, DIN (1989)
2. Colnarič, M., Verber, D., Gumzej, R., Halang, W.A.: Implementation of hard real-time embedded control systems. Real-Time Syst. **14**(3), 293–310 (1998). doi:10.1023/A:1007920407968. http://dx.doi.org/10.1023/A:1007920407968
3. Gumzej, R.: Holistic embedded control systems design with specification pearl. 1 CD-ROM. http://www.rts.uni-mb.si/misc/projekti/SPEARL/ (2006)
4. http://de.wikipedia.org/wiki/RTOS-UH
5. Pearl - process and experiment automation realtime language. http://www.pearl90.de/ (2014)
6. Gumzej, R., Lu, S.: Modeling distributed real-time applications with specification pearl. Real-Time Syst. **35**(3), 181–208 (2007)
7. 66253 Part 1: Basic pearl. Technical Report, DIN (1981)
8. 66253 Part 2: Full pearl. Technical Report, DIN (1982)

Springer

http://www.springer.com/978-3-319-28903-8

Engineering Safe and Secure Cyber-Physical Systems
The Specification PEARL Approach
Gumzej, R.
2016, XIII, 128 p. 28 illus., Hardcover
ISBN: 978-3-319-28903-8